Java, objetos e classes

Marco Dimas Gubitoso

26 de julho de 2013

Objetos

A ideia básica é construir uma abstração, o centro da programação não é mais voltada para como definir a sequência de operações (programação imperativa), mas como modelar melhor as informações e dados tratados pelo programa. A partir de um bom modelo, a programação fica mais fácil.

Este tipo de abordagem traz outras vantagens, como reutilização de código, extensibilidade e independência de implementação.

..continuação

O ponto de partida é o *objeto*, isto é, uma entidade representada no computador.

- Tudo é um objeto. Existem diversos *tipos*, ou *classes*, de objetos. Qualquer elemento de seu programa, incluindo o próprio, é representado como um objeto.
- Um programa é uma coleção de objetos interagindo entre si.
- Um objeto pode ser composto de outros objetos.
- Todo objeto tem um tipo. "Um objeto é uma instância de uma classe".
- Classes podem conter subclasses. (Um tipo é um conjunto, um subtipo é definido por restrições neste conjunto)
- Objetos da mesma classe reagem do mesmo modo.

..continuação

Uma forma mais simples de descrever um objeto é indicar três propriedades:

- Estado. Descrito por um conjunto de variáveis.
- Comportamento. Descrito por funções, normalmente chamadas de métodos no linguajar de orientação a objetos.
- Identidade. Indicada por uma instância (em última análise, um endereço na memória).

..continuação

Uma classe é representada no computador por uma *implementação*.

Um conjunto de variáveis descreve o estado (como em uma *struct*) e um conjunto de funções descreve o comportamento.

Encapsulamento

Para que o objeto A atue sobre o objeto B, não é necessário que ele conheça como B foi implementado. A única informação necessária é como chamar os métodos de B que fornecem as possíveis formas de interação. Na verdade, o implementador de A não deve fazer suposições sobre a implementação de B.

Toda classe pode esconder a implementação e apresentar apenas uma *interface*.

Herança

Uma classe pode ser apenas uma restrição sobre uma classe maior, um caso mais específico. Neste caso não há necessidade de se reconstruir toda a implementação: basta estender a implementação já existente da classe maior, restringindo o tipo.

Se B é uma classe derivada de A, B *usa a mesma interface* de A, talvez incluindo mais algumas coisas. O ponto importante é que em qualquer lugar do programa onde se usa um objeto do tipo A, pode-se também usar um objeto do tipo B sem a necessidade de reprogramação.

Existem duas formas de fazer herança: especializando (isto é, trocando métodos por outros mais específicos) ou estendendo (adicionando campos e métodos).

Polimorfismo

Funções que manipulam objetos de uma certa classe podem precisar atuar diferentemente se forem aplicadas a objetos de uma subclasse ou classes similares. Você, no programa, escreve a chamada do mesmo jeito para os dois casos, mas o código executado pode ser diferente.

Isto pode implicar em uma busca em tempo de execução (*late binding*), pois em algumas situações só é possível descobrir qual a função efetivamente chamada em tempo de execução.

Outra situação onde aparece o polimorfismo é em funções que recebem objetos como argumentos. Duas funções podem ter a mesma funcionalidade mas implementações diferentes, dependendo do tipo dos argumentos.

Implementando classes e objetos

Existem algumas diretrizes gerais para a implementação das classes. Estas diretrizes buscam manter o paradigma correto e garantir uma uniformidade na estrutura do código. Vamos discutir algumas delas, aproveitando para destacar alguns pontos importantes adicionais.

Agrupamento

Já vimos que classes podem ter uma relação hierárquica por meio de herança, mas esta não é a única possibilidade. Uma certa implementação pode precisar de classes auxiliares relacionadas e uma estrutura mais ampla do programa como um todo é interessante.

Assim, definições de classes podem ser agrupadas em "pacotes" (*packages*), cujos elementos possuem uma relação entre si, mas que são implementados com uma independência relativa.

Construtores

Toda classe tem um método especial para "criar" o objeto novo, na instanciação. Este método é chamado construtor e é responsável por alocar a memória para o novo objeto e fazer a sua inicialização.

É comum haver mais de uma forma de se criar um objeto e este é um dos casos onde o polimorfismo é usado com frequência.

Proteção e escopo

Para garantir o encapsulamento da implementação, a linguagem deve garantir meios para coibir o acesso à estrutura interna.

Os campos (variáveis e métodos) de uma classe podem ter três níveis de encapsulamento:

public Aberto. Qualquer método de qualquer classe pode ter acesso ao campo (manipular a variável ou chamar o método).

protected Protegida. Para ter acesso a este campo, o método precisa pertencer a uma subclasse ou a outra classe do mesmo pacote.

private Privado. Acesso permitido apenas a métodos da mesma classe. Isto exclui subclasses.

..continuação

De uma forma similar, as classes também podem ser encapsuladas em dois níveis:

public A classe pode ser instanciada ou estendida por qualquer outra classe.

Amigável Só pode ser usada dentro do pacote onde foi definida.

..continuação

Ainda no que diz respeito à abstração, uma classe pode especificar métodos que só serão definidos de fato em subclasses: são chamadas as classes abstratas. Não é possível instanciar objetos de classes abstratas. O outro extremo são classes que não permitem derivação. São classes "finais". Um método pode também ser final se ele não permitir nenhuma especialização (não puder ser sobrescrito.)

Da mesma forma, uma variável "final" é na verdade uma constante (não pode ser alterada) e um método abstrato não tem corpo.

Variáveis de classe

Existem situações onde é interessante ter uma variável na classe que é única para todas as instâncias, por exemplo, para se manter uma contagem de objetos já criados. Um método construtor pode usar uma variável destas para colocar um "número de série" nos objetos criados.

1 JAVA

Esta seção apresenta a linguagem Java e discute como os conceitos acima são implementados.

Tudo que se escreve em Java faz parte de algum objeto. Todas as classes são derivadas implicitamente de uma classe especial: `Object`. Esta derivação é implícita e não precisa ser especificada.

As únicas exceções, para efeitos de eficiência, são os tipos fundamentais. Se necessário, existem também uma classe para cada um destes tipos:

tipo	**tamanho**	**descrição**	**classe**
boolean	—	valor booleano `true` ou `false`	`Boolean`
char	16 bits	caractere (unicode)	`Character`
byte	8 bits	inteiro com sinal	`Byte`
short	16 bits	inteiro com sinal	`Short`
int	32 bits	inteiro com sinal	`Integer`
long	64 bits	inteiro com sinal	`Long`
float	32 bits	ponto flutuante (IEEE 754)	`Float`
double	64 bits	ponto flutuante (IEEE 754)	`Double`

1.1 Classes

Uma classe em Java é declarada pela cláusula `class`. A sintaxe é a seguinte:

> [*modificadores-de-classe*] `class` *nome* [`extends` *nome-da-superclasse*]
> [`implements` *lista-de-interfaces*] { *corpo* }

Os modificadores possíveis são os seguintes:

abstract Indica que a classe tem métodos abstratos (virtuais). Estes métodos também são marcados com este modificador.

final A classe não pode ter subclasses.

public A classe é pública. Só pode ter uma classe pública por arquivo.

Mais tarde falaremos de `extends` (herança) e `implements`.

O estado, como já vimos, é descrito por variáveis, declaradas de uma forma muito parecida com campos de uma `struct` em *C*. A principal diferença está nos modificadores `private`, `public`, `protect`, `final` e `static`, que já vimos.

Os métodos são declarados como as variáveis, mas com a possibilidade de se usar um modificador adicional: `abstract`. Um método `abstract` não tem corpo e deve ser definido em alguma subclasse. Apenas classes declaradas `abstract` podem ter métodos `abstract`.[1]

Um método com o mesmo nome da classe, sem tipo e público é chamado de *construtor*. Ele é chamado automaticamente quando a classe é instanciada.

---

[1] `interface`s são um caso especial, que veremos mais tarde.

1.2 Entrada e saída simples

O uso da saída padrão é feita por um objeto especial da classe `java.io.PrintStre` chamado `System.out`. Este objeto é `static` e oferece os métodos `print()`, `println()` e `flush()`.

A entrada padrão é manipulada por objetos da classes `java.io.InputStreamR` e `java.io.BufferedReader`.

1.3 Pacotes

Em Java definimos pacotes com a palavra reservada `package`.

No início do arquivo `MyClass.java`, pode-se usar

```
package mypackage;
public class MyClass {
  // . . .
```

Assim, `MyClass` pertencerá ao pacote `mypackage`. A diretiva `package` deve ser a primeira do arquivo.

Lembre-se que um arquivo `.java` só pode ter uma classe `public`, mas podem haver vários arquivos pertencentes ao mesmo pacote. Basta que todos usem o mesmo comando `package` no início e que estejam todos no mesmo diretório.

Esta restrição sobre a localização do arquivo tem a ver com a forma como Java armazena as classes compiladas, como veremos na próxima seção.

Existem duas formas de instanciar uma classe em outro pacote:

- Usando-se o nome completo:

```
mypackage.MyClass m = new mypackage.MyClass();
```

- Usando o `import`:

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

O mecanismo do `import` é muito conveniente pois permite usar bibliotecas de classes de uma forma bastante simples. Existem várias classes prontas e é muito fácil incluir novas.

1.4 Localização das classes

Ao compilar um programa, Java coloca cada classe em um arquivo `.class` diferente.

Os pacotes são colocados em diretórios, podendo haver subdiretórios para auxiliar na organização. Assim, se as classes de um certo pacote estão no subdiretório, digamos `242/aval/provas`, poderemos as classes deste pacote importanto da seguinte forma:

```
import 242.aval.provas.*
```

O diretório `242` deve estar incluído em um dos diretórios listados na variável de ambiente `CLASSPATH`.

Se se quiser importar apenas uma das classes deste pacote, usa-se o nome dela no lugar da `*`:

```
import 242.aval.provas.Nota
```

agora posso usar a classe `Nota` no programa.

Para garantir que não haverá confusão de nomes de pacotes, a convenção é usar o nome da `URL` invertido.

1.5 Herança

```
/**
 * Progressão numérica
 *
 * @author Do livro de Estruturas de dados
 */

public class Progression {
    /** Valor inicial */
    protected long first;

    /** Valor corrente */
    protected long cur;

    /** Construtor default */
    Progression() {
        cur = first = 0;
    }

    /** Reinicializa a progressão com o valor inicial
     *
     * @return Valor inicial
     */
```

```
/**
 * Progressão aritmética
 */

class Aritmética extends Progression {
    protected long inc;

    Aritmética() {
        this(1);
    }
    Aritmética(long incr) {
        inc = incr;
    }

    protected long próximoValor() {
        cur += inc;
        return cur;
    }
}
```

```
/**
 * Progressão aritmética
 */

class Aritmética extends Progression {
    protected long inc;

    Aritmética() {
        this(1);
    }
    Aritmética(long incr) {
        inc = incr;
    }

    protected long próximoValor() {
        cur += inc;
        return cur;
    }
}
```

```
/**
 * Progressão geométrica
 */

class Geométrica extends Progression {
    protected long base;

    Geométrica() {
        this(2);
    }
    Geométrica(long incr) {
        first = base = incr;
    }

    protected long próximoValor() {
        cur *= base;
        return cur;
    }
}
```

```
/**
 * Progressão de Fibonacci
 */

class Fibonacci extends Progression {
    protected long prev;

    Fibonacci() {
        this(0,1);
    }
    Fibonacci(long v1, long v2) {
        first = v1;
        prev = v2-v1;
    }

    protected long próximoValor() {
        long temp = prev;
        prev = cur;
        cur += temp;
        return cur;
    }
}
```

```
/**
 * Testador de progressões
 */

public class Tester {
    public static void main(String[] args) {
        System.out.println("Testador de progressões");
        Progression vp[] = {
            new Progression(),
            new Aritmética(5),
            new Geométrica(3),
            new Fibonacci(1,2)
        };
        for (int i = 0; i < vp.length; i++)
            vp[i].print(6);
    }
}
```

Figura 6: Testador

1.6 Interfaces

Para criar um objeto com características de dois tipos diferentes precisaríamos ter herança múltipla (a classe extenderia mais de uma classe base).

No entanto herança múltipla tem seus inconvenientes. Uma classe pode herdar implementações diferentes dos mesmos métodos ou ainda ser derivada mais de uma vez da mesma base. Existe ainda o problema de um campo ou método de um ancestral comum ser modificado indiretamente por métodos diferentes.

Vejamos um exemplo onde aparece a necessidade de herança múltipla: considere a hierarquia de alimentos da figura **??**.

Esta hierarquia pode ser facilmente estendida para vários casos. No entanto, temos um problema se precisarmos escolher alimentos de origem animal, mineral ou vegetal. Existem exemplos em diversos ramos desta árvore.

A solução, sem o fazer uso de herança múltipla, são as *interfaces*. Uma `interface` declara métodos que devem ser implementados em outras classes, como nos métodos virtuais. Instâncias de uma classe `C` que

implementa os métodos de uma interface `I` pode ser usado como objetos do tipo `I`, muito embora `I` não seja um tipo completo.

Este arranjo simples evita os problemas de herança múltipla mas fornece um mecanismo para que objetos sejam considerados como sendo de mais de um tipo, além do esquema de derivação usual.

No exemplo da figura **??**, podemos ter interfaces `Mineral`, `Animal` e `Vegetal`.

Além de métodos virtuais, uma interface pode ter campos (variáveis), mas elas obrigatoriamente são `static` e `final`.

1.7 (Contêineres)

Um *contêiner* é uma estrutura de dados que armazena uma coleção de objetos. Os exemplos mais simples de contêineres são vetores e matrizes (*arrays*). Outros contêineres são pilhas, filas, coleções, etc.

1.7.1 Arrays

Em Java os *arrays* são implementados de forma especial: são mais eficientes e são capazes de verificação de tipo. Esta implementação especial também possibilita o uso de '`[`' e '`]`' para referenciar um elemento.

Um *array* tem um tamanho fixo, determinado na sua criação, e não pode ser modificado dinamicamente. Além disso, o tipo dos elementos de um determinado *array* é conhecido e pode ser verificado — nos outros contêineres, o tipo é `Object`.

Java faz verificação de limites em **todos** os *arrays*. Um índice negativo ou além do tamanho do *array* provoca um erro de execução.

*Array*s também se comportam como objetos normais e têm um campo especial, `length`, que contém seu valor.

Um detalhe importante é que um *array*, quando criado, é inicializado com referências para *null*. Os elementos do *array* devem ser criados ou declarados explicitamente, a menos que seu tipo seja um dos tipos básicos[2].

[2]É sempre bom lembrar que todo objeto de um tipo não-básico é na verdade é uma referência

Em *java.utils* existe uma classe chamada `Arrays`, que possui uma série de métodos `static` que permitem operações sobre *arrays*:

- `fill(array, val)` — preenche um *array* com um valor.
- `sort(array)` — ordena um *array*.
- `equals(array1, array2)` — verifica igualdade.
- `binarySearch(array,valor)` — faz busca binária no *array*
- `asList(array)` — retorna o *array* transformado em uma lista (classe `List` do `java.utils`).

Existe também o método `static` do sistema, chamado `System.arraycopy()`, para copiar *arrays*.

1.7.2 Outros contêineres

Outros agrupamentos de objetos são implementados na biblioteca de Java2. Existem dois tipos adicionais de contêineres:

- Coleções (`Collection`), que podem ser listas (`List`) ou conjuntos (`Set`). Listas são coleções ordenadas e conjuntos são coleções que não permitem repetição.

- Vetores associativos (`Maps`). São associações do tipo (*chave*, *valor*). Tanto a chave como o valor pode ser um objeto qualquer (inclusive outros `Maps`).

Um grande problema com estes contêineres é que o tipo original do elemento se perde. Isto é, ao ser colocado no contêiner, o objeto é convertido para o tipo `Object`. Quando se recupera o objeto é necessária a conversão de volta para o tipo original.

Em Java não existem tipos parametrizados (`templates`); pelo menos por enquanto.

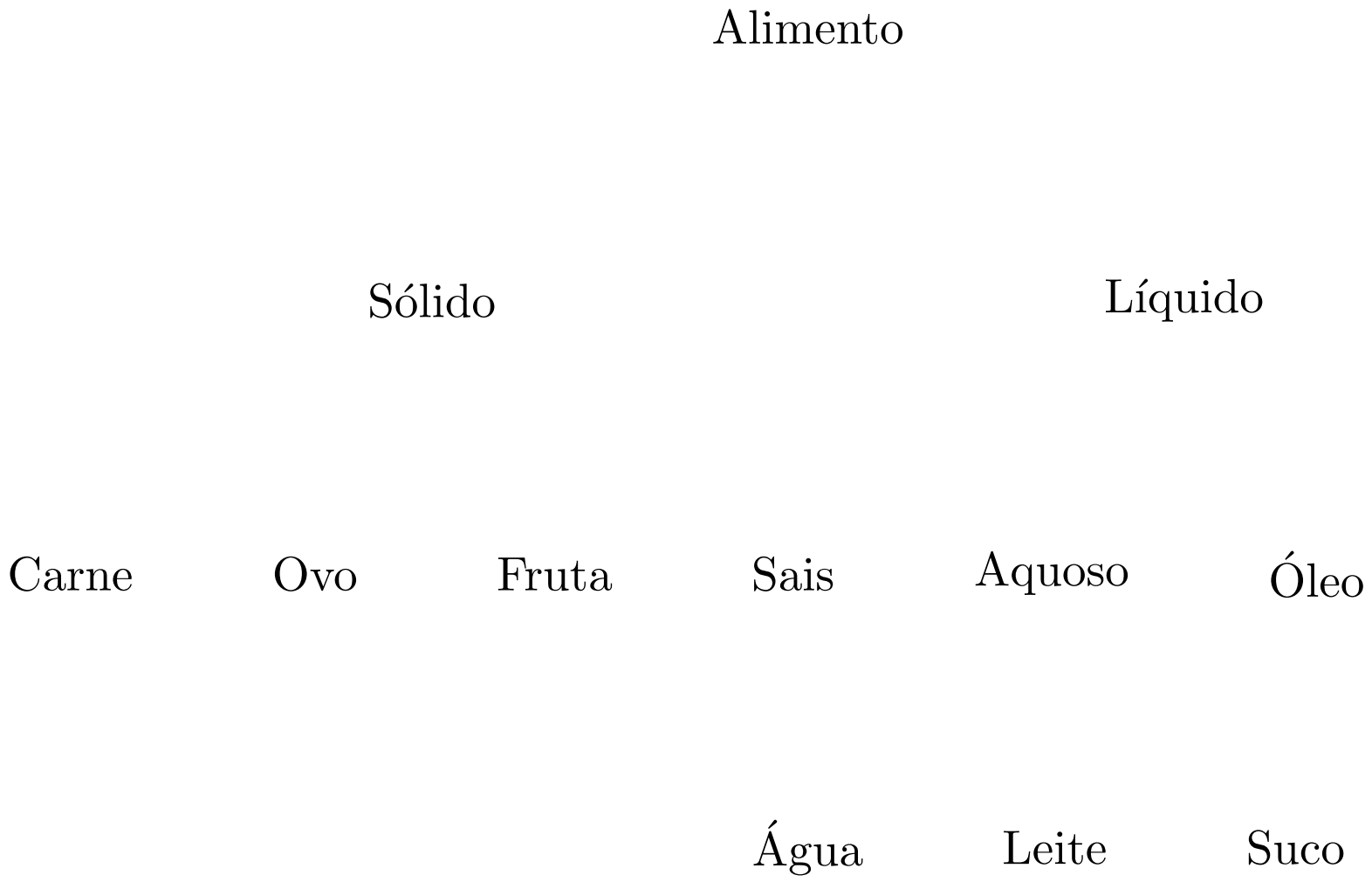

Figura 7: Hierarquia de Alimentos

Índice Remissivo